Programming Sensor Networks Made Easy *

Mahesh Arumugam and Sandeep S. Kulkarni

Software Engineering and Network Systems Laboratory
Department of Computer Science and Engineering
Michigan State University, East Lansing MI 48824
Email: {arumugam,sandeep}@cse.msu.edu
Web: http://www.cse.msu.edu/~{arumugam,sandeep}

Abstract. The designer of a sensor network protocol needs to address several *low-level* details such as message collisions, message losses, and resource limitations. Also, the designer needs to solve several *high-level* problems such as routing, leader election, and diffusing computation that are already considered in distributed systems and traditional networking. Therefore, to simplify the design of sensor network protocols, in this paper, we propose *ProSe*, a programming tool for sensor networks that enables the following: (i) specify programs in simple abstract models considered in distributed systems literature while hiding low-level details, (ii) reuse existing algorithms in the context of sensor networks, and (iii) automate code generation and deployment. Furthermore, ProSe helps in rapid prototyping and quick deployment of sensor networks. Finally, ProSe enables the transition where protocols are designed by *domain experts* rather than *experts in sensor networks*.

Keywords: programming abstraction, programming tool, code-generation, write-all-with-collision model, sensor networks

1 Introduction

Sensor networks have become popular recently due to their applications in unattended tracking and detection of undesirable objects, hazard detection, data gathering, environment monitoring, and so on. Due to their low cost and small size, it is easy to deploy them in large numbers. However, these sensors are constrained by limited power, limited memory and limited communication capability. Hence, they need to collaborate with each other to perform a certain task. Additionally, due to limited power, to sustain the network for longer duration, methods for power management are necessary. Also, since the sensors typically

* Addr: 3115 Engineering Building, Michigan State University, East Lansing MI 48824. Tel: +1-517-353-2387. Fax: +1-517-432-1061. This work was partially sponsored by NSF CAREER CCR-0092724, DARPA Grant OSURS01-C-1901, ONR Grant N00014-01-1-0744, NSF Equipment Grant EIA-0130724, and a grant from Michigan State University.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
2005		**00-00-2005 to 00-00-2005**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Programming Sensor Networks Made Easy	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Michigan State University,Department of Computer Science and Engineering,East Lansing,MI,48824**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**18**	

communicate over a single frequency, if multiple messages are sent to a sensor simultaneously then, due to collision, it receives none. For these reasons, the designer of a sensor network protocol needs to deal with several *mostly low-level* details such as message collision, limited memory, and limited power.

The designer of a sensor network protocol also has to solve several *high-level* problems that are considered in distributed systems and traditional networking. These include variations of consensus, routing, spanning tree construction, leader election, clock synchronization, and tracking. Therefore, to speed up the development and deployment of sensor networks, it is desirable to utilize the vast literature in this area in such a way that the low-level details of sensor networks are hidden from the designer.

One challenge in using existing algorithms is that the model considered in these algorithms does not account for the difficulties and opportunities provided by sensor networks. For example, in most existing sensors (e.g., Mica/XSM motes [1, 2]), the basic mode of communication is *local broadcast with collision*. In other words, when a sensor communicates, it can update the state of its neighbors. However, if a sensor receives 2 messages simultaneously then they collide and both messages become incomprehensible. Thus, the computation in sensor networks can be thought of as a *write all with collision* (WAC) model [3, 4]. By contrast, existing algorithms assume point-to-point communication and are mostly written in abstract models considered in distributed systems literature. These abstract models allow the designer to specify the program concisely by hiding several low-level details. (For a brief introduction of these abstract models, we refer the reader to Section 2.) Hence, to reuse existing programs (i.e., algorithms) in the context of sensor networks, these programs need to be transformed into programs in WAC model.

Contributions of the paper. With this motivation, we develop *ProSe*, a tool for programming sensor networks. The main features of ProSe are as follows.

- *Hide low-level details.* ProSe allows the designer of a sensor network protocol to specify programs in abstract models (e.g., shared-memory model, read/write model) considered in distributed systems literature. Programs written in such abstract models hide several low-level details, e.g., message collision, synchronization, resource limitations, etc. The transformation algorithm [3,4] used to transform a program into WAC model deals with such issues. Therefore, the designer can concentrate only on the functional aspects of the program. As a result, the development time of a sensor network protocol is significantly reduced.
- *Reuse existing algorithms.* ProSe allows the designer to reuse existing algorithms considered in the distributed systems literature. Since (variations of) problems such as consensus, routing, leader election, synchronization, and tracking are already addressed in distributed systems and traditional networking, reusing them in the context of sensor networks simplifies the design of sensor network protocols. Towards this end, ProSe enables rapid prototyping and quick deployment of sensor network protocols.

– *Automate code generation and deployment.* ProSe takes the input program written in an abstract model and automatically transforms it into a program in WAC model. Then, ProSe generates code for the transformed program that can be implemented on sensor networks. Currently, ProSe is targeted for nesC/TinyOS [5,6] platform. Once code is generated, ProSe uses the native tools of the platform to construct the binary image and then can disseminate the new binary across the network using a network programming service (e.g., [7–9]). Thus, ProSe allows the designer to automatically generate and deploy code.

Unlike manually designed protocols where the designer has to deal with several low-level details in addition to the functionalities of the protocol, ProSe allows the designer to rapidly prototype protocols and analyze the performance quickly. Additionally, if the transformation algorithm [3,4] preserves the properties of interest (e.g., fault-tolerance, stabilization [10,11]) of the original program, then ProSe also preserves such properties in the generated code.

Organization of the paper. The rest of the paper is organized as follows. In Section 2, we discuss how programs are specified in different computation models and briefly summarize the transformation algorithms proposed for WAC model. Then, in Section 3, we present an overview of the tool. Subsequently, in Section 4, we evaluate the performance of the tool. Specifically, we show that the programs generated by the tool provide comparable performance with respect to related programs designed manually for sensor networks. In Section 5, we discuss some of the questions raised by this work and in Section 6, we discuss the related work. Finally, in Section 7, we make the concluding remarks and present the scope for future research.

2 Preliminaries

In this section, we present the theoretical background on which our tool is based. We first precisely specify the structure of the programs written in shared-memory, read/write, or WAC models. Then, we discuss some of the results in transforming programs in shared-memory or read/write models into programs in WAC model.

2.1 Structure of Programs

The programs are specified in terms of guarded commands [12]; each guarded command (respectively, action) is of the form:

$$guard \quad \longrightarrow \quad statement,$$

where *guard* is a predicate over program variables, and *statement* updates program variables. An action $g \longrightarrow st$ is enabled when g evaluates to true and to execute that action, st is executed. A computation of this program consists

of a sequence $s_0, s_1, \ldots$, where s_{j+1} is obtained from s_j by executing actions in the program ($0 \leq j$).

A computation model limits the variables that an action can read and write. Towards this end, we split the program actions into a set of processes. Each action is associated with one of the processes in the program. We now describe how we model the restrictions imposed by shared-memory, read/write and WAC models.

Shared-memory model. In this model, in one atomic step, a sensor can read its state as well as the state of its neighbors and write its own (public and private) variables.

Read/Write model. In this model, in one atomic step, a sensor can either (1) read the state of one of its neighbors and update its *private* variables, or (2) write its own variables.

Write all with collision (WAC) model. In this model, each sensor consists of write actions (to be precise, write-all actions). Specifically, in one atomic action, a sensor can update its own state and the state of all its neighbors. However, if two or more sensors simultaneously try to update the state of a sensor, say k, then the state of k remains unchanged. Thus, this model captures the fact that a message sent by a sensor is broadcast. But, if multiple messages are sent to a sensor simultaneously then, due to collision, it receives none.

Remark. In this paper, we use the terms process and sensor interchangeably.

2.2 Transformations for WAC Model

The WAC model can be effectively used to model computations in sensor networks. Recently, approaches have been proposed for transforming programs into WAC model. They can be classified as: (a) TDMA based deterministic transformation [3] and (b) CSMA based probabilistic transformation [4].

TDMA based deterministic transformation. In [3], Kulkarni and Arumugam have proposed algorithms for transforming programs written in read/write model into programs in WAC model. In the transformations proposed in [3], the action by which a process (say, j) reads the state of process k in read/write model is modeled in WAC model by requiring process k to write the appropriate value at process j. However, if another neighbor of j is trying to write the state of j at the same time then, due to collision, none of the write actions succeed. In order to deal with this problem, in [3], time division multiple access (TDMA) is used to ensure that collisions do not occur during write actions.

In short, in WAC model, each process executes the actions for which the corresponding guard is enabled in the TDMA slots assigned to that process. And, each process writes (broadcasts) its state to all its neighbors in its TDMA slots. Furthermore, if the transformation uses a deterministic TDMA service to implement the write-all action, the resulting program in WAC model is also de-

terministic. Additionally, in [3], the authors propose extensions for transforming programs written in shared-memory model into programs in WAC model.

CSMA based probabilistic transformation. In [4], Herman proposed *cached sensor transform* (CST) that allows one to correctly simulate a program written for shared-memory model in sensor networks. CST uses CSMA to broadcast the state of a sensor and, hence, the transformed program is randomized.

We note that ProSe allows the designer of a sensor network protocol to use either of these transformations. Additionally, the developers of new transformation algorithm for WAC model can easily incorporate their algorithm in ProSe, thereby, enhancing the library of transformations available with ProSe.

3 ProSe: Overview

In this section, we present an overview of ProSe. Specifically, we discuss (1) the programming architecture of ProSe, (2) the input and output of ProSe, and (3) the execution sequence of ProSe.

3.1 Programming Architecture of ProSe

The programming architecture of ProSe is shown in Figure 1. ProSe transforms the input guarded command program into a program in WAC model (if the input program is specified in shared-memory or read/write model), as discussed in Section 2.2. Once the input guarded command program is transformed into WAC model, ProSe generates the corresponding nesC code (targeted for TinyOS). Furthermore, ProSe *wires* the generated code with a MAC layer (e.g., [13, 14]) to implement the write-all action in the WAC model. Such a service is useful in providing an interface for broadcasting (i.e., writing all neighbors) and receiving WAC messages.

Now, the designer of the sensor network protocol can use the TinyOS platform to build the binary of the nesC code. This binary image can then be disseminated across the network using a network programming service (e.g., [7–9]). Thus, sensor network protocols and applications can be specified and disseminated across the network.

3.2 Input/Output of ProSe

The input to ProSe consists of the guarded command program in shared-memory, read/write, or WAC model, its initial states and the topology of the network.

In the input guarded command program, the designer has to specify whether a variable is *public* or *private*. In shared-memory or read/write model, a sensor can read the public variables of its neighbors. Also, the designer has to identify the process (or sensor) to which the variable belongs. For example, if process j

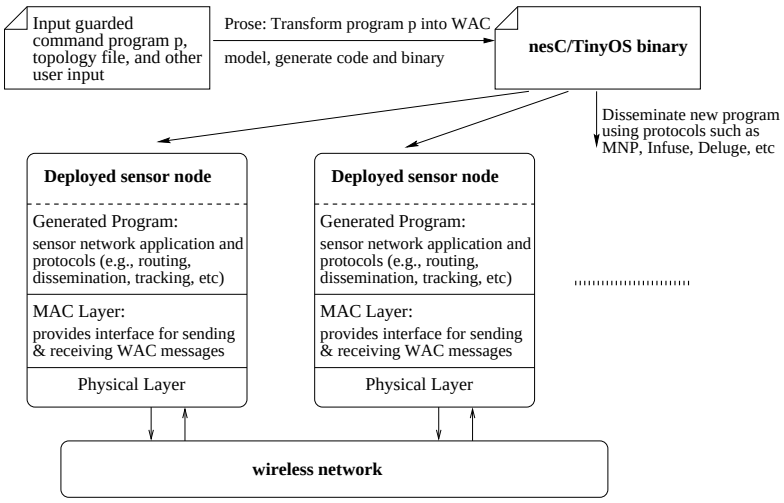


Fig. 1. ProSe programming architecture

accesses its local variable x, the designer has to specify it as $x.j$. Now, we discuss the input/output of ProSe in the context of an example.

Max program. In this program (MAX), each process (i.e., sensor) maintains a public variable x. The goal of the program is to eventually identify the maximum value of this variable across the network.

In MAX, whenever $x.j$ is less than $x.k$ (i.e., variable x at process k), j copies $x.k$ to $x.j$. This action allows j to update $x.j$ whenever its guard holds and, thus, j eventually computes the maximum value of x across the network. In the input file of ProSe, we specify the actions of j as follows (keywords are shown in *italics*):

```
1  program  max
2  process  j
3  var              public int  x.j;
4  begin
5          (x.k > x.j) -> x.j = x.k;
6  end
```

Initial state. The designer also specifies initial state in the input of ProSe. The initial state of the MAX program is as follows (*init* is a keyword):

```
1  init  x.j = j;
```

In the above example, $x.j$ is initialized to j (i.e., the ID of the sensor).

Topology file. Another input to ProSe is the topology file. This file identifies the ID of the base station, the size of the network, and the communication

topology (i.e., the neighborhood of each sensor). The communication topology is modeled as a directed graph. Each vertex represents a sensor node. Also, the topology identifies the packet reception rate (PRR) across each edge (or link). This can be determined using the information about the expected bit error rate across each edge and the network field characteristics. Additionally, PRR can be calculated from real data using a localization service (e.g., [15]) on the deployed network. Moreover, ProSe supports shortcuts for specific network topologies. For example, if the communication topology is a grid then it is sufficient to specify $GRID$ instead of listing all the edges. In the topology file, we specify these parameters as follows:

```
1 baseID: 1
2 size: 10
3 neighborhood: 1:2:0.95, 2:1:0.93, 1:3:0.91, 1:4:0.97, ...
```

In the above example, the base station ID is 1 and the size of the network is 10. The keyword *neighborhood* identifies the communication topology. The link $1:2:0.95$ denotes that sensor 1 can successfully communicate with sensor 2 95% of the time. Based on this information, the tool and the MAC layer used in the transformation compute the neighborhood of each sensor.

User interaction. Next, ProSe queries the user for the transformation algorithm and the MAC layer. Note that ProSe is not designed for a specific transformation algorithm or MAC layer. Rather, ProSe can be extended to work with different transformation algorithms and MAC layers. It maintains a library of transformation algorithms and MAC layer implementations. Based on the user input, it transforms the input program using the appropriate transformation and the MAC layer. The ProSe-user interaction is shown in Figure 2.

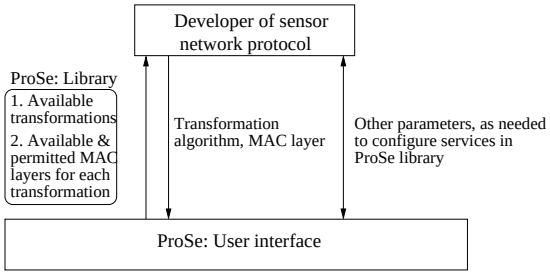


Fig. 2. ProSe-user interaction

If the user selects a TDMA based transformation (respectively, CSMA based transformation) then ProSe configures the TDMA (respectively, CSMA) service using the information provided in the topology file.

Output nesC program. ProSe generates the code for the input guarded command program in nesC as follows. As mentioned in Section 2.2, the read action in

shared-memory or read/write model is simulated in WAC model using the write-all action. Towards this end, each sensor maintains a copy vector for each public variable of its neighbor. This copy vector captures the value of the corresponding variable at its neighbors. The number of elements in this vector is determined using the information on the neighborhood of each sensor.

The actions of the input program are executed whenever a timer fires. Once the sensor executes each action for which the corresponding guard is enabled, it marshals all the public variables as a message *wacMsg* and schedules it for transmission (broadcast). Depending on the transformation algorithm and MAC layer selected by the user, it configures when the timer fires and how *wacMsg* is transmitted. For example, in case of a TDMA based transformation (e.g., [3]), ProSe configures the timer to fire in every TDMA slot assigned to the sensor. And, it uses a TDMA service (e.g., [13]) to broadcast the message. In case of a CSMA based transformation (e.g., [4]), ProSe configures the timer to fire in a random interval whenever it receives a message containing values of public variables at the sender. And, it uses a CSMA service (e.g., [14]) to broadcast *wacMsg*.

The nesC code segment for *Timer.fired()* event of the *MAX* program obtained using the TDMA based transformation from [3], where SS-TDMA service [13] is used to implement the write-all action, is shown as follows. (The function *getNoOfNbrs()* returns the number of neighbors of the given sensor.)

```
1  event result_t Timer.fired() {
2          uint8_t i, msgSizeInBytes;
3          if(sendDone == TRUE) {
4                 for(i = 0; i < getNoOfNbrs(); i++)
5                           if((copy_x[i] > x_j)) x_j = copy_x[i];
6                  wacMsg.data[0] = x_j; wacMsg.data[1] = (x_j >> 8);
7                  sendDone = FALSE;
8                  msgSizeInBytes = 2;
9                  call SSTDMA.send(msgSizeInBytes, &wacMsg);
10         }
11         return SUCCESS;
12 }
```

Similarly, ProSe generates code for updating the copy vectors whenever it receives a WAC message from one of its neighbors. Finally, ProSe also generates code for initialization. Specifically, it generates code to (1) initialize all the program variables, (2) configure network services (e.g., TDMA, CSMA), and (3) configure and start middleware services (e.g., Timer).

3.3 Execution of ProSe

In this section, we discuss (1) how the input guarded commands are translated into nesC code and (2) how the sensors maintain the state of their neighbors.

ProSe generates three files: *configuration* file, *module* file, and a *makefile*. Figure 3 shows how ProSe generates different files from the input program and topology information; these files are required to generate the TinyOS binary.

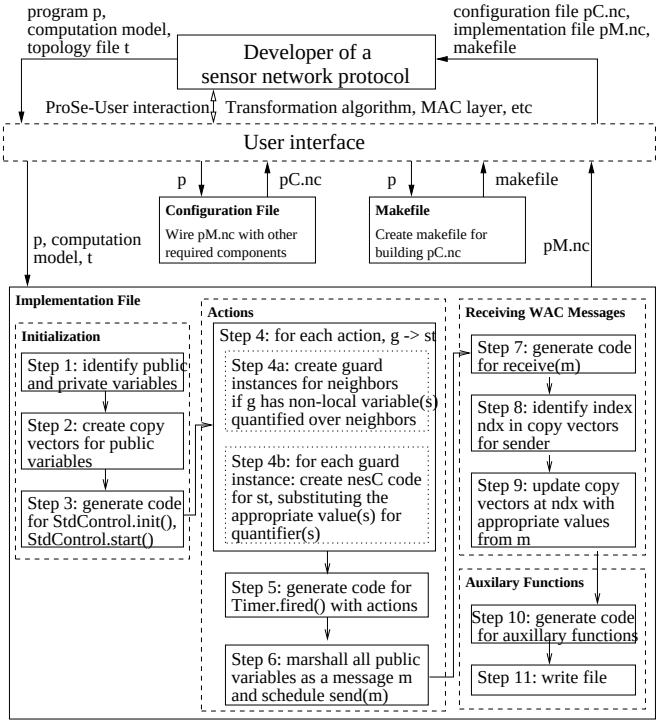


Fig. 3. Execution sequence of ProSe

Configuration file. Configurations wire components together, connecting interfaces used by components to interfaces provided by others [5]. ProSe generates *pC.nc*, given the input guarded command program *p*. Specifically, *pC.nc* wires the module file, *pM.nc*, network services (e.g., TDMA, CSMA, etc), and other interfaces required by the module.

Module file. Modules provide the application code and implement one or more interfaces [5]. ProSe generates the implementation file *pM.nc*, given the input guarded command program *p*, as follows (cf. Figure 3).

Steps 1-3: Initialization. First, ProSe identifies the public and private variables of the input program. Next, for each public variable, it generates a copy vector (containing entries for all the neighbors of a sensor). Subsequently, it generates the code for (i) initializing these variables, (ii) initializing other components (e.g., TDMA, CSMA, Timer, etc), and (iii) starting network and middleware services (e.g., TDMA, CSMA, Timer). In case of a TDMA based transformation, ProSe sets the Timer to fire during the TDMA slots assigned to the sensor. A sensor executes each action for which the corresponding guard is enabled whenever this timer fires.

Steps 4-6: Actions. ProSe generates the nesC code for the actions specified in the input program in *Timer.fired()* event. For each action $g \longrightarrow st$, first, it determines if the guard g has any non-local variables (e.g., process j accessing public variable $x.k$ of process k in the *MAX* program). If g contains no non-local variables, it generates the corresponding nesC code of the form $if(g)\{st;\}$.

If g has non-local variables, ProSe proceeds as follows. For each non-local variable, it generates guard instances for each neighbor. For example, in *MAX*, it generates *copy_x[0]* > *x_j*, *copy_x[1]* > *x_j*, *copy_x[2]* > *x_j*, and *copy_x[3]* > *x_j*, where j has 4 neighbors. If the index to a non-local variable is a local variable, ProSe generates a single guard instance for the neighbor identified by the index. For example, if g is of the form $x.(p.j) > x.j$, it generates *copy_x[getCopyVectorIndex(p_j)]* > *x_j*, where *getCopyVectorIndex(p_j)* identifies the index of *p_j* to the copy vectors. Now, for each guard instance g' (of g) with only local variables, ProSe generates the nesC code of the form $if(g')\{st;\}$.

Once the code for all actions are generated, ProSe generates code for implementing the write-all action. Towards this end, first, it marshals all public variables into a message. Then, it uses the MAC layer selected by the user to schedule transmission of the message.

Steps 7-9: Receiving WAC messages. ProSe generates code for updating the copy vectors whenever it receives a message. Towards this end, ProSe generates code for determining the sender of the message and the index of the sender to the copy vectors (using *getCopyVectorIndex(sender)*). Once the index is identified, the values of the public variables of the sender are updated in the corresponding copy vectors. Thus, each sensor maintains up-to-date values of the public variables of the neighbors. Furthermore, in case of CSMA based transformation, ProSe sets the timer to fire in a random interval whenever it receives a WAC

message. A sensor executes each action for which the corresponding guard is enabled whenever this timer fires.

Steps 10-11: Auxiliary functions. Finally, ProSe adds the code for all auxiliary functions (e.g., *getCopyVectorIndex(neighbor)*, *getNoOfNbrs()*, etc).

Makefile. To facilitate quick compilation and deployment, ProSe generates the makefile for building the TinyOS binary of the generated code.

Thus, ProSe provides the designer with the different files required for building and deploying the new binary image.

4 Evaluation: Manual Design Vs. Generated Programs

In this section, we study the performance of the programs generated by ProSe with respect to related programs designed manually for sensor networks, using an example. We consider a variation of balanced routing program [16]. We use ProSe to generate code for this program and experimentally analyze the performance of the generated program.

Modified version of balanced routing program. In this program, sensors are arranged in a logical grid. A routing tree is dynamically constructed with the base station as the root. The base station is located at $\langle 0, 0 \rangle$ of the logical grid. Each sensor classifies its neighbors as *high* or *low* neighbors depending on their (logical) distance to the base station. Also, each sensor maintains a variable, called *inversion count*. The inversion count of the base station is 0. If a sensor chooses one of its low neighbors as its parent, then it sets its inversion count to that of its parent. Otherwise, it sets its inversion count to inversion count of its parent + 1. Furthermore, to deal with the problem of cycles in the routing tree, if the inversion count exceeds a certain threshold ($CMAX$), the sensor removes itself from the tree.

In this program, each sensor (say, j) maintains three *public* variables: (i) $inv.j$, the inversion count of j, (ii) $dist.j$, the (logical) distance of j to the base station, and (iii) $up.j$, the status variable for j (indicates whether j has failed or not).[1] Whenever j finds a low/high neighbor that provides a better path (in terms of inversion count, $inv.j$) to the base station, it updates its *private* variable, $p.j$, the parent of j, and inversion count $inv.j$. The routing tree construction actions are specified in read/write model as follows. (Note that we present only the actions of the routing program below. For brevity, we drop the keywords *program*, *process*, variable declarations, *begin*, *end*, etc.)

```
1 Routing tree construction actions
2 ((dist.k < dist.j) && (up.k == TRUE) && (inv.k < CMAX)
```

[1] In this program, whenever a sensor fails, it notifies its neighbors. In practice, this action is implemented as follows. Whenever a sensor fails to read its neighbors (in shared-memory or read/write model) or receive update from its neighbors (in WAC model), for a threshold number of consecutive attempts, it declares that neighbor as failed. Thus, detecting whether sensors have failed is hidden from the designer.

```
3 && (inv.k < inv.j)) // low neighbor
4                         -> p.j = k; inv.j = inv.k;
5|
6 ((dist.k >= dist.j) && (up.k == TRUE) && (inv.k+1 < CMAX)
7 && (inv.k+1 < inv.j)) // high neighbor
8                         -> p.j = k; inv.j = inv.k+1;
```

Whenever a sensor fails or inversion count of a sensor exceeds $CMAX$, the routing tree changes. Towards this end, we need routing tree maintenance or stabilization actions. These actions are specified (in read/write model) as follows.

```
1 Routing tree maintenance or stabilization actions
2 // parent failed
3 ((p.j != NULL) && (up.(p.j) == FALSE))
4                         -> p.j = NULL; inv.j = CMAX;
5| // inversion count of p.j exceeds threshold
6 ((p.j != NULL) && (inv.(p.j) >= CMAX))
7                         -> p.j = NULL; inv.j = CMAX;
8| // inversion count is not consistent with p.j (low neighbor)
9 ((p.j != NULL) && (dist.(p.j) < dist.j) && (inv.j != inv.(p.j)))
10                        -> p.j = NULL; inv.j = CMAX;
11| // inversion count is not consistent with p.j (high neighbor)
12 ((p.j != NULL) && (dist.(p.j) >= dist.j) && (inv.j != inv.(p.j)+1))
13                        -> p.j = NULL; inv.j = CMAX;
14| // j is not in tree and inversion count is not CMAX
15 ((p.j == NULL) && (inv.j < CMAX))
16                        -> p.j = NULL; inv.j = CMAX;
```

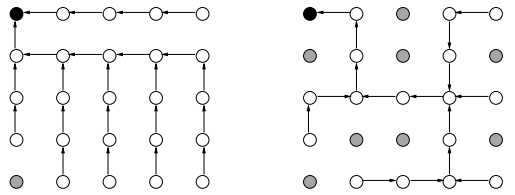
Thus, the routing program is specified in read/write model while hiding low-level details. Now, we can use ProSe to generate the corresponding nesC/TinyOS implementation and subsequently construct the binary image.

Experimental evaluation of program generated by ProSe. We use the TDMA based transformation from [3] to transform the above routing program into WAC model. Towards this end, we use SS-TDMA [13] to implement the write-all action. We deployed the code generated by ProSe on 3x3 and 5x5 XSM [2] networks, where the inter-sensor separation is 8 ft. In our experiments, the period between successive slots assigned to a sensor is 1.95 seconds. We kept this value intentionally high in order to reduce the frequency of execution of actions. After the initial routing tree is constructed, we fail some sensors (simultaneously) to determine how the sensors converge to a new routing tree and measure the convergence time. In case of a 3x3 (respectively, 5x5) network, we fail 2 (respectively, 7) sensors. The results of these experiments are presented in Table 1.

Figure 4 shows the initial routing tree and the converged routing tree (after sensors fail) on a 5x5 XSM network, where one of the sensors fail at the start of the experiment. After the initial routing tree is constructed (cf. Figure 4(a)), we fail sensors 3, 6, 8, 10, 17, 18, and 20 simultaneously. The sensors then converge to the new routing tree (cf. Figure 4(b)) within 21 seconds.

Table 1. Performance of routing program generated by ProSe

Network size	No. of failed sensors	Convergence time
3x3	2	4 s
5x5	7	21 s



(a) **(b)**

Fig. 4. Routing tree construction and maintenance on a 5x5 network with base station (filled circle) at the top-left corner. (a) initial routing tree and (b) converged tree after failure of some sensors (shown in gray circles)

The convergence time of the routing program generated by ProSe is within the acceptable performance guidelines of a typical sensor network application (e.g., [17, 18]). Also, the above results are comparable to routing programs designed manually for sensors networks (e.g., [19]). We experimentally computed the convergence time with MintRoute [19] and found it to be approximately the same. In such manually designed programs, the designer has to deal with all the low-level details of sensor networks. However, in the programs generated by ProSe, the transformation algorithm and the tool deal with these issues and, hence, the designer can focus only on the functionality of the program.

Based on this illustration, we expect that the programs generated by ProSe are comparable with respect to related programs designed manually. In order to quickly prototype applications, it is preferable that the designers use this tool to test existing algorithms in the context of sensor networks. If the performance of the prototype generated by ProSe meets the application requirements, the designer can choose to use the generated prototype. Otherwise, the designer can improve this prototype instead of designing from scratch. Thus, our tool enables the transition where protocols are designed by *domain experts* rather than *experts in sensor networks*. With this feature, designers can significantly reduce the development time of a typical sensor network application.

5 Discussion

In this section, we discuss some of the questions raised by this work.

Other services for sensor networks. We have used ProSe to generate sensor network binaries for (1) *network services* such as routing (e.g., [16]), diffusing computation (e.g., [20]), leader election (e.g., [21, 22]) and spanning tree construction (e.g., [11, 21]), (2) *distributed reset service* [23] to reset the state of the network to a consistent global state, and (3) *tracking service* to monitor the activities of mobile targets in a sensor network field (e.g., [24]). These programs are specified in abstract models (e.g., shared-memory model, read/write model). ProSe transformed them into WAC model and generated the corresponding nesC/TinyOS code. Thus, ProSe can be effectively used in automating the process of code generation and deployment of services for sensor networks.

Preserving fault-tolerance/self-stabilization properties. Properties such as fault-tolerance and self-stabilization are important in sensor networks. Specifically, since sensor networks are deployed in large numbers and in inaccessible fields, the network should be able to *self-stabilize* [10, 11] in presence of faults (e.g., message corruption, message losses, synchronization errors, etc). Towards this end, ProSe preserves the fault-tolerance and self-stabilization properties of the program in WAC model. Additionally, if the algorithm used in transforming a program (in shared-memory or read/write model) into a program in WAC model preserves the properties of interest then ProSe also preserves such properties. Existing transformations [3, 4] preserve the fault-tolerance and self-stabilization properties of the programs in shared-memory or read/write models.

Dealing with faults in sensor networks. The normal operation of a typical sensor network is affected by (1) failure of sensors, (2) state corruption, and (3) message losses. To deal with these problems, ProSe provides abstractions to the designer of a sensor network protocol. First, ProSe provides the abstraction which allows a sensor (say, j) to determine whether its neighbor (say, k) is alive or failed. Towards this end, in the input program, sensor j can just access the public variable $up.k$; if $up.k$ is *true* (respectively, *false*) then k is alive (respectively, failed). However, in the code generated by ProSe, the logic for determining whether a sensor is alive or not is as follows. If j fails to receive update messages (i.e., WAC messages) for a pre-determined time interval from its neighbor k, then j declares k as failed. Thus, the designer can abstract sensor failures using the up variables (cf. Section 4 for an example).

As discussed earlier, we note that ProSe preserves the properties of the input program in WAC model. If the original program *self-stabilizes* [10, 11] to legitimate states (from state corruption) and the transformation preserves this property then the code generated by ProSe also preserves this property. Moreover, with this approach, the designers can model malicious sensors. Finally, regarding message losses, the transformation algorithm should ensure that each sensor updates their state at their neighbors every pre-determined time interval. This ensures that the sensors have the current values of the public variables of their neighbors. Thus, ProSe deals with faults in sensor networks.

6 Related Work

Related work that deals with programming abstractions include [25–28] and tools for programming sensor networks include [29–36].

Programming abstractions. In [25], a state centric approach is proposed that captures algorithms such as sensor fusion, signal processing and control. This model views the network as a distributed platform for in-network processing. Furthermore, in this model, the abstraction of *collaboration groups* hides the designer from issues such as communication protocols, event handling, etc. In [26, 27], *macroprogramming* primitives that abstract communication, data sharing and gathering operations are proposed. These primitives are exposed in a high-level language. However, these primitives are application-specific (e.g., *abstract regions* for tracking and gathering [26] and *region streams* for aggregation [27]). In [35], an intermediate language (called, *token machine language*) for programming sensor networks using these primitives is proposed. Unlike the state centric programming model [25] and the macroprogramming primitives [26, 27, 35], ProSe only hides low-level details such as message collisions, message corruption, sensor failures, etc. As a result, with ProSe, the designer has more freedom to develop network protocols (e.g., clustering, leader election, routing, etc) that provide the desired results. Additionally, ProSe facilitates rapid prototyping by allowing designers to reuse existing algorithms.

In [28], *semantic services* programming model is proposed where each service provides semantic interpretation of the raw sensor data or data provided by other semantic services. In this model, users only specify the end goal on what semantic data to collect. Thus, users make less low-level decisions on which operations to run or which data to run them over. However, semantic services model does not capture all sensor network protocols. For example, network protocols such as routing, clustering, leader election, etc, do not fit in this model. Additionally, users are limited by the services available in the network. By contrast, ProSe is generic, i.e., it can be used to specify wide variety of sensor network protocols (e.g., routing, leader election, distributed reset, tracking, etc). Moreover, unlike [28], ProSe does not restrict the designer to implement the desired services.

Programming tools. In [29–34], virtual machine, database, or middleware based programming models are proposed. Specifically, (1) in [29], a virtual machine based approach (called *Maté*) is proposed for programming and adapting sensor network applications, (2) in [30], an object-based distributed middleware service (called *EnviroTrack*) with interfaces to application developer is proposed, especially for environment monitoring, (3) in [31], a sensor network application construction kit (*SNACK*) that includes a configuration language and a library of services is proposed to simplify construction of sensor network applications, (4) in [32], a self-explanatory, easy to configure/maintain interface (called *TASK*) to sensor network deployment is proposed, (5) in [33], database query-link interface, called *TinyDB*, is proposed for designing sensor network applications, and (6) in [34], mobile agents are used to specify application tasks.

However, the approaches in [29–34] are (i) application-specific, and/or (ii) restrict the designer to what is available in the virtual machine, middleware, library, or network. By contrast, ProSe provides a simple abstraction while allowing the designer to specify wide variety of protocols.

In [36], macroprogramming model, called *Kairos*, that hides the details of code-generation and instantiation, data management, and control is proposed. Kairos provides the programmer with three abstractions; (i) node-level abstraction that allows the programmer to manipulate nodes and list of nodes, (ii) one-hop neighbor list abstraction for performing operations on the neighbor list, and (iii) remote data access that allows a sensor to read the named sensors. Thus, it allows one to specify the desired global behavior in a centralized model. While ProSe provides similar abstractions, it differs from Kairos. Specifically, unlike Kairos, ProSe only hides low-level details such as message collisions, corruption, sensor failures, etc. Moreover, ProSe does not require any runtime support in the generated sensor network binary. Additionally, unlike Kairos, ProSe enables reuse of existing algorithms and also preservers fault-tolerance properties of the input program.

7 Conclusion

In this paper, we presented a tool, called *ProSe*, for programming sensor networks. ProSe simplifies the construction and deployment of sensor network protocols. Specifically, ProSe (1) hides the low-level details (e.g., message collisions, sensor failures, limited memory, etc) of sensor networks from the designer, thereby enabling the designer to focus only on the functionality of the protocol, (2) allows reuse of existing algorithms considered in distributed systems and traditional networking in the context of sensor networks, (3) preserves the properties (e.g., self-stabilization, fault-tolerance) of the program in WAC model, (4) reduces the development time of a sensor network protocol, thereby enabling the designer to rapidly prototype and quickly deploy sensor network applications, and (5) automates the process of code generation and deployment. Furthermore, ProSe is extensible in that developers of transformation algorithms and MAC protocols can easily integrate new transformations and communication services in ProSe without a significant overhead.

We expect that the performance of programs generated by ProSe are comparable with respect to related programs designed manually. Hence, it is preferable that the designers use ProSe to test existing algorithms in sensor networks. If the performance of prototype generated by ProSe meets application requirements, the designer can choose to use the prototype. Otherwise, the designer can improve the prototype generated by ProSe instead of designing from scratch. Thus, ProSe enables the transition where protocols are designed by *domain experts* rather than *experts in sensor networks*. We demonstrated this by generating nesC/TinyOS code for the balanced routing program [16] specified in read/write model. Furthermore, we have generated sensor network binaries for network protocols (e.g., [20–22]), distributed reset [23], and tracking [24].

There are several possible future directions to this work. While ProSe can generate programs that can be deployed on a sensor network, it is expected that sensor network applications be composed of several protocols. Towards this end, one future direction is to extend ProSe to compose several protocols together. Specifically, ProSe should prioritize the public variables that are transmitted (i.e., broadcasted) and ensure that the composition is correct. Another future direction is to integrate ProSe with (i) tools that automatically synthesize fault-tolerance properties (e.g., [37]) and (ii) tools that formally verify (input and transformed) programs (e.g., model checkers).

References

1. J. Hill and D. Culler. Mica: A wireless platform for deeply embedded networks. *IEEE Micro*, 22(6):12–24, 2002.
2. P. Dutta, M. Grimmer, A. Arora, S. Bibyk, and D. Culler. Design of a wireless sensor network platform for detection rare, random, and ephemeral events. *In Proceedings of the Conference on Information Processing in Sensor Networks*, 2005.
3. S. S. Kulkarni and M. Arumugam. Transformations for write-all-with-collision model. *Computer Communications (Elsevier)*, 2005, in press. Available at: `http://www.cse.msu.edu/~sandeep/publications/ka05COMCOM/`.
4. T. Herman. Models of self-stabilization and sensor networks. *In Proceedings of the 5th International Workshop on Distributed Computing (IWDC)*, 2003.
5. D. Gay, P. Levis, R. voh Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. *In Proceedings of Programming Language Design and Implementation (PLDI)*, June 2003.
6. TinyOS: A component-based OS for the networked sensor regime. `http://www.tinyos.net`.
7. S. S. Kulkarni and L. Wang. MNP: Multihop network reprogramming service for sensor networks. *In Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, June 2005.
8. S. S. Kulkarni and M. Arumugam. Infuse: A TDMA based data dissemination protocol for sensor networks. Technical Report MSU-CSE-04-46, Department of Computer Science, Michigan State University, November 2004.
9. J. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. *In Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys)*, November 2004.
10. E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11), 1974.
11. S. Dolev. *Self-Stabilization*. The MIT Press, 2000.
12. E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, 1997.
13. S. S. Kulkarni and M. Arumugam. SS-TDMA: A self-stabilizing MAC for sensor networks. In *Sensor Network Operations*. IEEE Press, 2005.
14. A. Woo and D. Culler. A transmission control scheme for media access in sensor networks. *International Conference on Mobile Computing and Networking*, 2001.
15. T. He, C. Huang, B. Blum, J. Stankovic, and T. Abdelzaher. Range-free localization schemes for large scale sensor networks. *In Proceedings of the International Conference on Mobile Computing and Networking (MobiCom)*, pages 81–95, 2003.
16. J. A. Cobb and M. G. Gouda. Balanced routing. *In Proceedings of the International Confernece on Network Protocols (ICNP)*, pages 277–284, October 1997.

17. A. Arora et al. A line in the sand: A wireless sensor network for target detection, classification, and tracking. *Computer Networks (Elsevier)*, 2004.
18. A. Arora et al. ExScal: Elements of an extreme scale wireless sensor network. *In Proceedings of the International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, August 2005.
19. A. Woo and D. Culler. Taming the challenges of reliable multihop routing in sensor networks. *ACM Conference on Embedded Networked Sensor Systems*, 2003.
20. A. Arora, M. Gouda, and G. Varghese. Constraint satisfaction as a basis for designing nonmasking fault-tolerance. *Journal of High Speed Networks*, 1996.
21. N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.
22. S. Dolev, A. Israeli, and S. Moran. Uniform dynamic self-stabilizing leader election. *IEEE Transactions on Parallel and Distributed Systems*, 8(4):424–440, April 1997.
23. A. Arora and M. Gouda. Distributed reset. *IEEE Transactions on Computers*, 43(9):1026–1038, 1994.
24. M. Demirbas, A. Arora, and M. Gouda. A pursuer-evader game for sensor networks. *In Proceedings of the Symposium on Self-Stabilizing Systems*, June 2003.
25. J. Liu, M. Chu, J. Liu, J. Reich, and F. Zhao. State-centric programming for sensor-actuator network systems. *Pervasive Computing*, 2003.
26. M. Welsh and G. Mainland. Programming sensor networks using abstract regions. *Symposium on Networked Systems Design and Implementation (NSDI)*, 2004.
27. R. Newton and M. Welsh. Region streams: Functional macroprogramming for sensor networks. *Workshop on Data Management for Sensor Networks*, 2004.
28. K. Whitehouse, F. Zhao, and J. Liu. Semantic streams: A framework for declarative queries and automatic data interpretation. Technical Report MSR-TR-2005-45, Microsoft Research, April 2005.
29. P. Levis and D. Culler. Maté: A tiny virtual machine for sensor networks. *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.
30. T. Abdelzaher et al. EnviroTrack: Towards an environmental computing paradigm for distributed sensor networks. *In Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, 2004.
31. B. Greenstein, E. Kohler, and D. Estrin. A sensor network application construction kit (snack). *ACM Conference on Embedded Networked Sensor Systems*, 2004.
32. P. Buonadonna, D. Gay, J. Hellerstein, W. Hong, and S. Madden. TASK: Sensor network in a box. *European Workshop on Wireless Sensor Networks*, 2005.
33. S. Madden, M. Franklin, J. Hellerstein, and W. Hong. TinyDB: An acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems (TODS)*, 30(1):122–173, 2005.
34. C-L. Fok, G-C. Roman, and C. Lu. Rapid development and flexible deployment of adaptive wireless sensor network applications. *International Conference on Distributed Computing Systems*, 2005.
35. R. Newton, Arvind, and M. Welsh. Building up to macroprogramming: An intermediate language for sensor networks. *In Proceedings of the International Conference on Information Processing in Sensor Networks (IPSN)*, April 2005.
36. R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using kairos. *In Proceedings of the International Confernece on Distributed Computing in Sensor Systems (DCOSS)*, June-July 2005.
37. S. S. Kulkarni and A. Ebnenasir. A framework for automatic synthesis of fault-tolerance. Technical Report MSU-CSE-03-16, Department of Computer Science, Michigan State University, July 2003.